Project Task: Automated Code Correction

Introduction

This project introduces a next-generation, agentic APR system that leverages recent advances in large language models (LLMs) and multi-agent coordination to perform intelligent, explainable, and adaptive software repair. Built using the LangGraph and LangChain frameworks, the system adopts an agentic architecture, enabling it to decompose complex repair tasks into modular steps—such as defect categorization, code understanding, patch generation, fix explanation, and validation—each handled by specialized agents.The system is tested on the QuixBugs benchmark suite, covering 40 diverse algorithmic programs and 14 well-defined defect types. It integrates with state-of-the-art LLMs, including Gemini, to drive reasoning and code synthesis, ensuring both high repair accuracy and interpretability.

Thought Process

The objective of this task was to design and implement an automated system capable of repairing buggy programs from the **QuixBugs** dataset. The system needed to perform the following sequential steps:

1. **Ingest Buggy Code**: Accept source code files from the QuixBugs dataset known to contain logical or syntactic defects.
2. **Analyze the Code**: Understand the structure and behavior of the buggy code, identify the defect category (e.g., off-by-one, incorrect operator, etc.), and generate a semantic understanding of the error.
3. **Generate a Fix**: Use an LLM-driven agent to propose a corrected version of the code based on the identified defect and contextual understanding.
4. **Validate the Fix**: Test the fixed code against the expected outputs defined in the tester.py files included in the QuixBugs suite, ensuring behavioral correctness.

Blockers

Single Agent Design

Initially, I attempted to implement the entire repair workflow using a single agent. However, this approach quickly became impractical due to overly long and complex prompts, which led to reduced clarity, context overflow, and inconsistent outputs from the language model.

Buggy Code Interpretation

A major challenge was enabling the agent to reason about buggy code. Since LLMs can struggle with understanding faulty logic in isolation, creating an agent exclusively to interpret the function of the buggy code also by utilizing and extracting comments.

Error 429

To handle 429 "Too Many Requests" errors from the LLM API, I implemented a retry mechanism with exponential backoff. The invoke_with_retry function retries failed requests up to 5 times. If a 429 error is detected, it waits increasingly longer before retrying (e.g., 5s, 10s, 20s...), giving the server time to reset limits. For other errors, it stops immediately.

Validation Strategy with tester.py

Understanding how to correctly validate the repaired code using the provided tester.py files in the QuixBugs dataset was initially unclear. I encountered issues like **infinite loops** and other runtime errors. To resolve this, I had to make **minor modifications** to some tester.py files to prevent such errors and ensure the repaired code could be tested correctly. Also had to add timeout conditions in case correct python and bad python were stuck in infinite loops.

Approach

State Management with AgentState

The workflow begins with the definition of a TypedDict called AgentState to maintain and track the state across different stages of the repair pipeline. This structured state contains fields like the program name, buggy code, defect category, understanding, fixed code, validation result, test cases, and reference code for comparison.T his structured and typed state ensures **clear data flow**, **consistency**, and **easy debugging** across agents.

Tool Definition for Core Tasks

LangChain was used to define modular, reusable tools for key tasks such as:

- **Fetching Buggy Code** from the QuixBugs dataset
- **Fetching Reference Code** to compare against the fixed version
- **Comparing Outputs** between the fixed and reference code for both syntactic and logical similarity

Multi-Agent Workflow with LangChain Agents

The system is built using a chain of agents, each responsible for a specific task:

- **Agent 1: Defect Categorization** — Identifies the type of bug using structured prompts.
- **Agent 2: Code Understanding** — Extracts high-level summaries and comments to interpret the buggy code.
- **Agent 3: Code Repair** —Generates a fix based on the defect category and the code insights produced by Agent 2. Initially, the idea was to map specific repair strategies to each defect type, but further analysis revealed that a small subset of defects occurred frequently. This led to a more targeted repair mapping for those common defect categories. Further, also explain the bug and the fix that the agent has provided.
- **Agent 4: Validation** — Tests the fixed code and compares its output against the reference to ensure correctness. A few modifications were made to the tester.py file so that infinite loops were avoided.

Key Features

Smart Prompting with Defect Generalization

Instead of mapping a repair strategy to every possible defect—which can make prompts too long—selected the most common defect categories and included only generalized repair examples for these in the agent's prompt. This keeps the prompt concise and focused. The prompt also reminds the model not to rely solely on these examples, encouraging flexible reasoning for rare or novel defects.

Enhanced Code Understanding via Comments

To improve contextual understanding of the buggy code, the agent actively utilizes any comments present in the source code. During the code understanding phase, the agent's prompt instructs it to make use of comments to infer the intended behavior and structure of the program

Error Fixing with Explanation

Unlike typical automated repair agents that only output the corrected code, my agent not only fixes the identified error but also provides a concise explanation for the fix. This dual output—code plus rationale—ensures that users understand both *what* was changed and *why* it was necessary, enhancing transparency and educational value. The explanation is generated as part of the agent's workflow and is explicitly separated from the code output.

Results

- Total Programs Processed: 40
- Successful Fixes: 38 (95.0%)
- Failed Fixes: 2

Future Enhancements

Multi-Agent Collaboration Framework for Multilingual Instruction Generation

We propose a multi-agent collaboration framework where specialized agents—each an expert in a different programming language—work together to generate high-quality multilingual instruction data. Each agent first creates language-specific instructions from code snippets, then collaborates with others to synthesize instructions and solutions that are broadly applicable across languages. (1)

Agentic Feedback Loop for Increasing Similarity

To enhance similarity in recommendation pipelines, the Agentic Feedback Loop (AFL) framework uses two agents—a recommendation agent and a user agent—that iteratively refine recommendations and feedback. The recommendation agent suggests items with rationales, while the user agent simulates feedback and reasons. Both use memory of past interactions to update their strategies through multiple rounds, aligning recommendations more closely with user preferences. This process leads to higher-quality, user-aligned recommendations and improved accuracy.

Implementation

Buggy python program	Corrected python program	Fixed python program

```
def bitcount(n):
    count = 0
    while n:
        n ^= n - 1
        count += 1
    return count
```

```
def bitcount(n):
    count = 0
    while n:
        n &= n - 1
        count += 1
    return count
```

```
def bitcount(n):
    count = 0
    while n:
        n &= n - 1
        count += 1
    return count
```

```
def bucketsort(arr, k):
    counts = [0] * k
    for x in arr:
        counts[x] += 1

    sorted_arr = []
    for i, count in enumerate(arr):
        sorted_arr.extend([i] * count)

    return sorted_arr
```

```
def bucketsort(arr, k):
    counts = [0] * k
    for x in arr:
        counts[x] += 1

    sorted_arr = []
    for i, count in enumerate(counts):
        sorted_arr.extend([i] * count)

    return sorted_arr
```

```
def bucketsort(arr, k):
    counts = [0] * k
    for x in arr:
        counts[x] += 1

    sorted_arr = []
    for i, count in enumerate(counts):
        sorted_arr.extend([i] * count)

    return sorted_arr
```

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(a % b, b)
```

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

Appendix

Code Modification in tester.py file

```python
def run_py_try_with_timeout(algo, test_in, correct=False, fixed=False, timeout=3):
    py_code = f"""
import copy
import sys
from tester import py_try, prettyprint
algo = '{algo}'
test_in = {test_in}
correct = {correct}
fixed = {fixed}
try:
    result = py_try(algo, *copy.deepcopy(test_in), correct=correct, fixed=fixed)
    print(prettyprint(result))
except Exception as e:
    print('ERROR:', e)
"""

    try:
        completed = subprocess.run(
            [sys.executable, "-c", py_code],
            capture_output=True,
            text=True,
            timeout=timeout
        )
        return completed.stdout.strip()
    except subprocess.TimeoutExpired:
        return "TIMEOUT"
```

References:

https://jkoppel.github.io/QuixBugs/quixbugs.pdf

https://arxiv.org/pdf/1805.03454

https://arxiv.org/pdf/2505.10468