SP-202 Agentic Software Debugger & Documenter: Software Requirements Specification

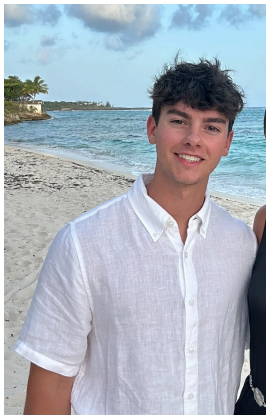
CS 4850 — Sec 02 — Spring 2026

January 27, 2026

Sharon Perry

Jade Le

Team Lead
Documentation & Testing



Preston Dietz

Development & Testing

Omodemi Olaoluwa

Development & Testing

Team Members:

Name	Role	Cell Phone / Alt Email
Jade Le (Team Lead)	Documentation & Test	818.270.8979 jle12@students.kennesaw.edu
Preston Dietz	Development & Test	678-997-7713 pdietz1@students.kennesaw.edu
Olaoluwa Omodemi	Development & Test	470-266-9850 oomodem2@students.kennesaw.edu
Sharon Perry	Project Owner / Advisor	770.329.3895 Sperry46@kennesaw.edu

Table of Contents

1. Introduction

1.1. Overview

Our agentic software debugger and documenter is a multi-agent system, created with Google's ADK, that analyzes source code autonomously to identify bugs, provide fixes, and create documentation. It will be centered on self-directed decision-making, tool usage, and self-correction, with both autonomous and rule-based decision-making processes. It will be implemented as a command-line interface program that produces trace logs to keep the decision-making process transparent.

1.2. Purpose

The project's purpose is to demonstrate the application of agentic AI in software development processes, such as debugging and documenting source code written in Python. This system will be able to:

- Demonstrate the use of a multi-agent system pipeline in the allocation of specialized roles such as bug detection, fixes, and documentation.
- Develop an iterative system of fixes, validation, and refinement that increases the quality of the output over multiple steps while keeping it bounded.
- Generate trace logs that enable instructors and developers to observe the decision-making processes, not just the end result.

1.3. Intended Users

Students and developers who want quick assistance identifying and correcting common issues in Python code.

1.3.1. User Assumptions

- Users are assumed to be able to run a CLI (command-line interface) program and provide a Python file.
- Users are assumed to be able to interpret the output such as lint errors, patch-style changes, and Markdown documentation.

2. Context & Motivation

2.1. Limitations of Traditional Tools

Traditional debugging tools and static analyzers are valuable but limited:

- Debuggers (breakpoints/stepping) reveal program state but not root cause or recommended resolution order.
- Linters/static analyzers catch syntax/style and some unsafe patterns but typically lack semantic reasoning about programmer intent.
- These tools do not coordinate multi-step workflows.

2.2. Limitations of Deterministic Software

A deterministic script can apply rules, but debudding often requires judgement under uncertainty:

- The optimal fix depends on context and intent that may not be explicitly encoded.
- When information is missing, deterministic systems usually fail instead of proposing reasonable candidates and validating them.

2.3. Agentic Hypothesis

A multi-agent approach, where specialized agents iteratively propose and validate fixes, will improve debugging workflow quality compared to a single-pass deterministic tool, while trace logs and iteration limits keep autonomy observable and bounded.

3. System Scope

3.1. Included Features

This system will:

- Accept a Python source code file as input via CLI.
- Run a multi-agent pipeline:
    1. Bug detection/issue identification
    2. Fix proposal and code modification
    3. Validation using a linter

4. Iterative refinement if validation fails
5. Documentation generation for final output

- Output:
    - A corrected code version (even if not fully lint-clean after max iterations).
    - Linter feedback from the final validation attempt.
    - Generated Markdown documentation (README-style).
    - Structured trace logs for the full session.

3.2. Excluded Features

This system will NOT:

- Create entirely new, unrelated programs from scratch (it only modifies provided code).
- Execute user code with network access or external database access.
- Modify files outside the provided input file(s).
- Provide real-time collaborative or IDE-integrated debugging (CLI only).
- Run unbounded loops (must terminate after max iterations).

3.3. Assumptions & Constraints

3.3.1. Assumptions

- Input is Python source code and is readable as a text file.
- The system has access to a configured Python runtime environment sufficient to run a linter locally (no network required).
- The user provides code that is within reasonable size limits for CLI processing (e.g., not an entire repository).
- The project environment permits use of Google ADK patterns (SequentialAgent / LoopAgent).

3.3.2. Constraints

- Python only
- No network execution or external database connections.
- Max 3 refinement iterations per input.
- CLI-based interface only.

- Every execution session must produce a trace log that includes agent actions and tool calls.
- System must terminate after the iteration cap and return the best available output.

4. Functional Requirements

4.1. The multi-agent system shall support Python source code as input.

4.2. The multi-agent system shall autonomously identify potential bugs.

4.3. The multi-agent system shall attempt code corrections using tools.

4.4. The multi-agent system shall validate corrected code using a linter.

4.5. Based on linter results, the multi-agent system shall either trigger a refinement iteration (on errors) or proceed to documentation generation (on success).

4.6. The multi-agent system shall limit the number of refinement iterations to a maximum of three attempts per input.

4.7. The multi-agent system shall terminate the refinement loop and return the best available output if the maximum iteration limit is reached.

4.8. The multi-agent system shall generate documentation from final output.

4.9. The multi-agent system shall generate structured trace logs for each execution session, including agent actions and tool calls.

5. Non-Functional Requirements

5.1. The multi-agent system shall provide clear CLI prompts/help text indicating required inputs, run status (iteration count), and where outputs are written.

5.2. The multi-agent system shall produce trace logs in a structured format (e.g. JSON) that can be parsed programmatically for evaluation.

5.3. The multi-agent system shall always terminate after at most 3 refinement iterations and shall not run indefinitely.

5.4. The multi-agent system shall not execute code with network access and shall not access external databases or modify files outside the provided input.

5.5. The multi-agent system's workflow shall be modular (separate agents for detection, fixing, documentation) to support future replacement or extension of individual agents without redesigning the full pipeline.