SP-202 Agentic Software Debugger & Documenter: Development Document

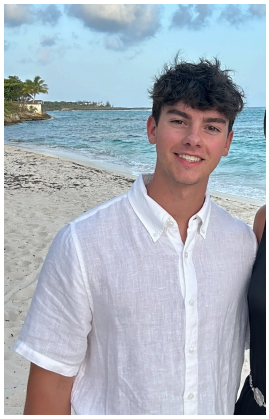
CS 4850 — Sec 02 — Spring 2026

March 1, 2026

Sharon Perry

Jade Le

Team Lead
Documentation & Testing



Preston Dietz

Development & Testing

Olaoluwa Omodemi

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. Technical Narrative

1.1. Introduction

This document describes our current development process for our agentic software debugger and documenter. It details how to set up our development environment, how we are building our agents, workflow orchestration, and validation logic.

1.2. Agent Implementation

1.2.1. Base Agent - Baseagent.py

Baseagent.py serves as the foundation class that all other agents inherit from. It handles all shared initialization steps: loading the API key from the .env file, instantiating the Gemini model via the Google ADK, creating the ADK runner, and establishing a session context. By centralizing these setup routines in one place, each specialized agent only needs to define its own prompt logic and output handling without duplicating boilerplate code.

1.2.2. Bug Detection Agent - Detector.py

The Detector agent is responsible for analyzing a given Python source file and identifying code-level issues. Its prompt is structured to instruct the model to examine the code for common bug categories, including logic errors, type mismatches, unsafe defaults, and style violations.

Input: The agent receives the raw text content of a Python file.

Output: The agent returns a structured JSON object listing each detected issue. Each issue includes a line number, a category label (e.g., MUTABLE_DEFAULT, BARE_EXCEPT, NONE_COMPARISON), and a plain-language description of the problem

1.2.3. Fixer Agent and Iterative Loop - Fixer.py and Loop.py

Prompt Design: fix only what the bug report flags, nothing more, and always return output in a structured format with the fixed code in a labeled code block.

How corrections are generated: FixerEngine first applies fast rule-based fixes (mutable defaults, bare excepts, None comparisons) with no API call, then the LLM handles any remaining issues that require reasoning.

LoopAgent / Retries: The pipeline passes an iteration counter into fix_file() each cycle. After each fix the linter runs, and if errors remain the pipeline calls fix_file() again with iteration + 1.

Maximum iteration logic / Termination: MAX_ITERATIONS = 3 is enforced at the top of fix_file(). If iteration > 3 it immediately returns MAX_ITERATIONS_REACHED with the best available code without calling the LLM.

1.2.4. Documentation Agent - Documentation.py

The Documentation agent converts the final corrected code into structured markdown documentation. It generates clear sections including code summaries, detected issues, applied fixes, and explanations of changes. Markdown was chosen for its readability and compatibility with common documentation tools such as GitHub. The agent enforces consistent formatting to ensure the output is easy to interpret and reusable.

1.3. Linter and Iteration Workflow

Validation is integrated into the fix-and-retry loop to ensure each iteration of the Fixer agent produces measurable improvement. The linter acts as the objective quality gate that determines whether an additional fix cycle is needed.

1.3.1. Linter Integration

Linting is executed after every call to fix_file(). The validator module (validator.py) runs the linter against the current version of the corrected code and captures its output, including all error codes, line numbers, and descriptions. These results are then passed back into the pipeline as a structured list, allowing the Fixer agent to receive targeted feedback on which issues remain unresolved in the next iteration.

1.3.2. Iteration Trigger Logic

A retry is triggered whenever the linter reports one or more remaining errors after a fix attempt. The pipeline compares the current error count to the previous count; if errors have decreased, the loop continues. If the error count does not decrease between iterations If MAX_ITERATIONS is reached, the system returns the most recent version of the code with a status flag indicating incomplete resolution. The caller (pipeline.py) logs this outcome and proceeds to the Documentation agent using whatever code is available at that point.

1.4. Development Challenges

A key challenge encountered during development was API rate limiting when using the free tier of the Google Gemini API. The system relies on multiple agent calls per pipeline execution (Detector, Fixer, and Documentation agents), and the iterative fix loop can trigger repeated API requests in quick succession. This often

caused requests to be throttled or temporarily blocked, interrupting the workflow.

1.5. Lessons Learned

Separation of concerns between agents makes individual components easier to test and debug. Keeping the Detector, Fixer, and Documentation agents independent allowed the team to iterate on each without breaking the others.

2. Data and Connectivity

2.1. Database

Not applicable to our project. All intermediate state (bug reports, fix iterations, linter output) is held in memory during a single pipeline run and is not stored between sessions.

2.2. Connectivity

Our system connects to Google Gemini API through the Google Agent Development Kit (ADK). All agent reasoning, tool invocation, and structured outputs are generated through authenticated API calls to Gemini models. This connectivity layer enables the agents to operate autonomously and exchange structured JSON responses.

3. Setup and Deployment Guide

3.1. Requirements

- Python 3.10+
- Google ADK
- Google Gemini API key

3.2. Installation

1. Install Python 3.10+
2. Clone the repository:
   ```
   git clone https://github.com/Agentic-debugger/Debugger
   ```
3. Install dependencies:
   ```
   pip install -r requirements.txt
   ```
4. Create a .env file in the root directory and add your Google Gemini API key:
   ```
   GOOGLE_API_KEY = your_api_key_here
   ```
5. Create virtual environment and activate:

```
uv venv --python 3.12
.venv\Scripts\activate
```

3.3. Project Structure

```
Debugger/
├── Agents/
│   ├── Baseagent.py
│   ├── Detector.py
│   ├── Documentation.py
│   ├── Fixer.py
│   └── Loop.py
│
├── Linting/
│   ├── rules.py
│   └── validator.py
│
├── Orchestration/
│   ├── controller.py
│   ├── pipeline.py
│   └── state.py
│
├── Documentation/
│   ├── SP-202 - Red - Agentic SWD - REQUIREMENTS.pdf
│   ├── SP-202 - Red - Agentic SWD - DESIGN.pdf
│   └── SP-202 Red Agentic SWD Project Plan.pdf
│
├── main.py
├── README.md
├── requirements.txt
└── .gitignore
```